

A GPU-based parallel algorithm for enumerating all chordless cycles in graphs

Walid A. R. Jradi Elisângela S. Dias Diane Castonguay Humberto Longo
Hugo A. D. do Nascimento

Universidade Federal de Goiás

{walid.jradi,elisangela,diane,longo,hadn}@inf.ufg.br

June 26, 2015

Abstract

In a finite undirected simple graph, a *chordless cycle* is an induced subgraph which is a cycle. We propose a GPU parallel algorithm for enumerating all chordless cycles of such a graph. The algorithm, implemented in OpenCL, is based on a previous sequential algorithm developed by the current authors for the same problem. It uses a more compact data structure for solution representation which is suitable for the memory-size limitation of a GPU. Moreover, for graphs with a sufficiently large amount of chordless cycles, the algorithm presents a significant improvement in execution time that outperforms the sequential method.

Keywords: Graphs, Chordless Cycles, Parallel Algorithm, GPU, OpenCL.

1 Introduction

Consider a finite undirected simple graph $G = (V, E)$, with $n = |V|$ and $m = |E|$. A *chordless cycle* is an induced subgraph that is a cycle, i.e., there is no edge outside the cycle connecting two vertices of it.

Sequential and parallel algorithms to the problem of determining if a graph contains a chordless cycle with $k \geq 4$ vertices, for some fixed cycle length k , were proposed by Chandrasekharan et al. [4]. They presented an algorithm, where a cycle C_l , $l \geq k$ can be found in $\mathcal{O}(m^2 \cdot n^{k-4})$ time sequentially and in $\mathcal{O}(\log n)$ time using $\mathcal{O}(m^2 \cdot n^{k-4})$ processors in parallel on a CRCW PRAM. However, finding just one cycle of length greater or equal to a fixed value k is easier than enumerating all chordless cycles in a graph G .

In general, enumeration is classified as belonging to the class of \mathcal{P} -complete problems, whose resolution is comparatively as hard as the resolution of problems in \mathcal{NP} -complete class [3, 23]. Although there are sequential algorithms to solve problems in such class, they become impractical as the problem size grows, preventing its utilization and requiring the usage of other approaches, like heuristics and meta-heuristics (trying to find a good enough approximate solution) or parallel computing (aiming the reduction of the algorithm's execution time).

Many sequential algorithms have been proposed for enumerating graph structures such as cycles [1, 7, 13, 14, 16, 17, 24], circuits [2, 20], paths [9, 16], trees [11, 16] and cliques [15, 21]. This kind of tasks is usually hard to deal with, since even a small graph may contain a huge number of such structures. Nevertheless, enumeration is necessary in the resolution process of many practical problems. In particular, the enumeration of chordless cycles is useful in areas like the study of ecological networks with the aim of discovering the predators that compete for the same prey [19]. Usually, a directed *food web* graph is transformed into a *niche overlap* graph to represent the competition between species, see [25]. The lack of chordless cycles in the later graph means that the species can be rearranged along a single hierarchy. Another application

of enumeration of chordless cycles is the prediction of nuclear magnetic resonance chemical shift values [18].

A sequential algorithm that enumerates all chordless cycles is described by Sokhn et al. [19]. The general principle of this algorithm is to use a vertex ordering and to expand paths from each vertex using a depth-first search (DFS) strategy. This approach has the disadvantage of finding twice each chordless cycle. Unfortunately, the authors did not present a complexity analysis of the algorithm.

Another sequential algorithm to enumerate chordless cycles was proposed by Uno and Satoh [22] and, as the algorithm of Sokhn et al. [19], each chordless cycle will appear more than once in the output. Actually, each cycle will appear as many times as its length. Thus, the algorithm has $\mathcal{O}(n \cdot (n + m))$ time complexity in size of the sum of lengths of all the chordless cycles in the graph.

Dias et al. [6] developed, up to our knowledge, the fastest sequential algorithm to enumerate all chordless cycles in undirected graphs since it finds all cycles just once. It is recursive and based on a depth-first search (DFS) strategy, with $\mathcal{O}(n + m)$ time complexity in the output size. Although the technique presented in [6] surpasses all solutions currently available, it still takes a considerable processing time when applied to some complex graphs or to graphs with a large amount of chordless cycles.

Again, up to our knowledge, there is no parallel algorithm for the problem of enumerating all chordless cycles in an undirected graph. In this paper, we fill in this gap by presenting a GPU-based parallel algorithm to such problem that is fast when applied to complex graphs and/or large amount of chordless cycles.

The remaining of this paper is organized as follows. In Section 2, we establish some preliminary definitions. In Section 3, we present the idea of our sequential algorithm. The parallel algorithm is introduced in Section 4. Section 5 describes the experimental tests and the results produced by the new algorithm compared against other methods. Conclusion and future work are discussed in Section 6.

2 Background

In this section, we present some mathematical definitions that support our approach to enumerate all chordless cycles of a graph. For more details on these definitions, see [6].

Let $G = (V, E)$ be a finite undirected simple graph with vertex set V and edge set E . Let $n = |V|$ and $m = |E|$. We denote by $Adj(x) = \{y \in V \mid (x, y) \in E\}$ the set of neighbors of a vertex $x \in V$ and by $Adj[x] = \{x\} \cup Adj(x)$ the closed neighborhood of x .

A *simple path* is a finite sequence of vertices $\langle v_1, v_2, \dots, v_k \rangle$ such that $(v_i, v_{i+1}) \in E$ and no vertex appears repeated in the sequence, that is, $v_i \neq v_j$, for $i, j \in \{1, \dots, k-1\}$ and $i \neq j$. A *cycle* is a simple path $\langle v_1, v_2, \dots, v_k \rangle$ such that $(v_k, v_1) \in E$. We denote a cycle with k vertices by C_k ¹. A *chord* of a path (resp. cycle) is an edge between two vertices of the path (cycle), that is not part of it. A path (cycle) without chord is called a *chordless path* (*chordless cycle*).

The minimum degree among all vertices of G is denoted by $\delta(G)$. The maximum degree is denoted by $\Delta(G)$; for reason of simplicity, we use just Δ . We represent by $d_G(v)$ the degree of a particular vertex $v \in V$. The subgraph induced by the subset $V - X$, for $X \subseteq V$ ($V - \{u\}$, for $u \in V$), is denoted by $G - X$ ($G - u$).

An ordering of the vertices of G can be defined by a bijection $\ell : V \rightarrow \{1, 2, \dots, n\}$. We call such a bijection a *vertex labeling*. Given a such vertex labeling, if

1. G contains a simple cycle $\langle v_1, v_2, \dots, v_k \rangle$,
2. $\ell(v_2) = \min\{\ell(v_i) \mid i = 1, \dots, k\}$ and

¹Note that our definition of cycle does not repeat the first vertex at the end of the sequence as usually done by other authors. We decided to use this definition (with the first vertex implicitly included at the end) because it simplifies the representation of a rotated version of the cycles. If $\langle v_1, v_2, \dots, v_k \rangle$ is a cycle, so also are $\langle v_i, v_{i+1} \dots, v_k, v_1, v_2, \dots, v_{i-1} \rangle$ and $\langle v_i, v_{i-1}, \dots, v_2, v_1, v_k, \dots, v_{i+1} \rangle$, for all $i = 1, \dots, k$.

3. $\ell(v_1) < \ell(v_3)$,

then ℓ defines the cycle in a unique way. Note that any cycle can be described as $\langle v_i, \dots, v_k, v_1, v_2, \dots, v_{i-1} \rangle$ or $\langle v_i, \dots, v_2, v_1, v_k, \dots, v_{i+1} \rangle$, for all $i = 1, \dots, k$. Let i be a vertex index such that $\ell(v_i) = \min\{\ell(v_j) \mid j = 1, \dots, k\}$. There are only two possibilities for the vertex v_i to be the second one of the cycle: $\langle v_{i-1}, v_i, v_{i+1}, \dots, v_k, v_1, v_2, \dots, v_{i-2} \rangle$ or $\langle v_{i+1}, v_i, v_{i-1}, \dots, v_2, v_1, v_k, \dots, v_{i+2} \rangle$. Since the neighbors of v_i in the cycle are v_{i-1} and v_{i+1} , exactly one of these possibilities satisfies the condition 3.

In the approach introduced in [6], a vertex labeling is given by a particular bijection $\ell : V(G) \rightarrow \{1, \dots, n\}$ called *degree labeling*. It is constructed over a sequence of subgraphs of G , starting with $G_1 = G$. For $i \geq 1$, the $(i+1)^{\text{th}}$ subgraph is defined as $G_{i+1} = G_i - u_i$, for a chosen $u_i \in V(G_i)$ such that $d_{G_i}(u_i) = \delta(G_i)$. Given such a sequence, the degree labeling is defined as $\ell(u_i) = i$ for each i .

A *triplet* is defined as a sequence of vertices that can initiate a chordless path of length greater than three. Let $T(G)$ denote the set of all initial valid triplets of G , that is, $T(G) = \{\langle x, u, y \rangle \mid x, u, y \in V \text{ with } x, y \in \text{Adj}(u), \ell(u) < \ell(x) < \ell(y) \text{ and } (x, y) \notin E\}$. The above labeling scheme allows to find every chordless cycle only once and to begin with a smaller initial set of chordless paths, which reduce considerably the search space. Note that, for any chosen degree labeling, if G is a tree then there are no possible triplets, that is, $T(G) = \emptyset$. Moreover, if G has a unique cycle then $|T(G)| = 1$, no matter what degree labeling is used, that is, unneeded triplets are discarded. As detailed in [6], an upper bound for the initial search space size is given by $|T(G)| \leq \frac{(\Delta-1) \cdot m}{2}$.

Given a chordless path $p = \langle v_1, v_2, \dots, v_k \rangle$ and a vertex $v \in \text{Adj}(v_k)$ such that $v \neq v_{k-1}$, thus exactly one of the following occurs:

1. $\langle p, v \rangle = \langle v_1, v_2, \dots, v_k, v \rangle$ is a chordless path;
2. there exists $i \in \{1, \dots, k-1\}$ such that $p = \langle v_i, v_{i+1}, \dots, v_k, v \rangle$ is a chordless cycle.

Since $v \in \text{Adj}(v_k)$, $v \neq v_{k-1}$ and p is a chordless path, then $\langle p, v \rangle$ is a simple path. Suppose that $\langle p, v \rangle$ is not a chordless path. Therefore, there is an index $i \in \{1, \dots, k-1\}$ with $(v, v_i) \in E$. Choosing the biggest index i with this property, we have the desired chordless cycle. Case 1 states that path $\langle p, v \rangle$ is an expansible chordless path. Case 2, with $i \neq 1$, state that path $\langle p, v \rangle$ has a chord or, with $i = 1$, $\langle p, v \rangle$, is a chordless cycle.

3 The sequential algorithm

The sequential algorithm for chordless cycles enumeration of Dias et al. [6] is briefly described here in order to help the understanding of the proposed, parallel approach. Further details and experimental results can be found in [6]. A pseudo-code of this algorithm is presented in Algorithm 1.

A degree labeling is initially calculated for the input graph G (Line 1). Then, the set of initial valid triplets $T(G)$ (Line 2) is computed, as described previously in Section 2. The set C is initialized (Line 3) with all triangles (which are also chordless) and variable T receives the set $T(G)$ (Line 4). Next, for each triplet $t = \langle x, u, y \rangle \in T$, a DFS strategy is used to check the existence of a chordless cycle starting at its last vertex (y) and respecting the constraints on the labeling order. Line 9 of the algorithm verifies if the addition of a neighbor of y to the path gives:

1. a chordless cycle;
2. a chord in the current path; or
3. another expansible path.

In case 1, the newly chordless cycle found is added to the set C of cycles (Line 11); in case 2, the path is discarded and, in the last case, the expanded path is added to the set T of expandable paths (Line 13). The same process is repeated until the set T becomes empty.

Due to the initial conditions of the triplets and the way the search is performed, the algorithm finds all chordless cycles and still avoids rotations of the same solution (two or more cycles with the same structure but that start at different vertices). This provides a faster execution of the algorithm. Dias et al. [6] presented another version of the algorithm, that uses a specialized breadth-first search (BFS), to ensure that each path expansion finds a chordless cycle. However, in practice, the algorithm without BFS leads to a shorter execution time.

Algorithm 1: *SequentialChordlessCycles(G)*

Input: Graph G .

Output: Set C of all chordless cycles of G .

```

1 perform DegreeLabeling( $G$ )
2  $T(G) \leftarrow \{\langle x, u, y \rangle \mid x, u, y \in V : x, y \in Adj(u); \ell(u) < \ell(x) < \ell(y) \text{ and } (x, y) \notin E\}$ 
3  $C \leftarrow \{\langle x, u, y \rangle \mid x, u, y \in V : x, y \in Adj(u); \ell(u) < \ell(x) < \ell(y) \text{ and } (x, y) \in E\}$ 
4  $T \leftarrow T(G)$ 

5 while ( $T \neq \emptyset$ ) do
6    $p \leftarrow \langle v_1, v_2, \dots, v_t \rangle \in T$ 
7    $T \leftarrow T - \{p\}$ 
8   foreach  $v \in Adj(v_t)$  do
9     if ( $(\ell(v) > \ell(v_2))$  and ( $v \notin Adj(v_i), i \in \{2, \dots, t-1\}$ )) then
10      if  $v \in Adj(v_1)$  then
11         $C \leftarrow C \cup \{\langle p, v \rangle\}$ 
12      else
13         $T \leftarrow T \cup \{\langle p, v \rangle\}$ 

14 return  $C$ .
```

A possible strategy for the parallelization of Algorithm 1 is the extension of multiple chordless paths through the simultaneous checking of the feasibility of adding each one of the neighbors of the last vertex on each path. The following sections detail this approach.

4 A GPU-based parallel algorithm

In this section, we present our parallel approach for the chordless cycles enumeration problem. The strategy adopted for parallelizing the computation done by Algorithm 1 is to split it into two stages and define a parallel approach for each one of them. The first stage involves the creation of the set C , with all cycles of length three, and the set $T(G)$, with all initial valid triplets $\langle x, u, y \rangle$ (Lines 2–4 of Algorithm 1). The second stage gets each path $\langle x, u, y, \dots, v \rangle$ in $T(G)$, that characterizes a chordless path, and tries to extend it by adding a neighbor to the last vertex, v (Lines 5–13).

The computation of a degree labeling (that appears at Line 1 of Algorithm 1), however, was not parallelized. Due to its inherent sequential nature and to the low impact in the processing time of the algorithm, this step was kept sequential as a preprocessing task and the resultant labels were used in our parallel stages.

The final parallel algorithm was mapped to a GPU architecture, which basic concepts are presented just below.

4.1 GPU Programming

General-Purpose programming on Graphics Processing Units (GP-GPU) technique consists in the utilization of GPUs to run non-graphical applications. GPUs are *stream processors* – that is, execution units able to operate in parallel, simultaneously performing routines in a large amount of data. They are focused on data parallelism and the basic idea is to maximize the data flow rate rather than to minimize latency (as in CPUs). GPU architecture emphasizes the implementation of many “light” threads concurrently, instead of executing a few traditional heavy threads.

In this work, it is adopted a very common GPU architecture consisting of p Symmetric Multiprocessors (SM) and a large, but slow, global memory. These processors, in turn, are grouped into larger units called *Symmetric Multiprocessors* (SM). Each SM, therefore, has a subset of $\frac{p}{|SM|}$ processors, where $|SM|$ is the number of SM units. All processors within a SM can communicate through a small, but fast, local shared memory. Processors in different SMs can only communicate through the global memory. There is also a private memory, unique to each running thread, which is much faster but smaller than the other memories.

Developing efficient data structures using the GPU memory model is a challenging task by itself [12]. The distinct characteristics of GPU and CPU architectures make many ordinary data structures (as the ones used in the sequential algorithm described in [6]) not suitable for parallelization in GPUs. Thus, other data structures and data flows have to be employed to overcome limitations (such as small memory size) and to take advantages of the characteristics of the different types of the GPU memories.

Next, problems with the data structures of the sequential algorithm are discussed and new data structures for the parallel algorithm are presented. After that, the following section details the two parallel stages mentioned above.

4.2 Data structures

Usually, graphs are represented by adjacency matrices or lists. Although an adjacency matrix allows verifying connectivity between two vertices in constant time, it has three primary issues:

- for sparse graphs, there is a significant waste of memory space;
- due to the large space occupied, it is not possible to allocate the entire matrix in the fast, but small, GPU local shared memory. Even in advanced GPU models, this memory does not exceed 64KB. A simple graph containing just 256 vertices would be enough to fill in all this memory ($256 \cdot 256 \cdot 1 \text{ byte} = 65536 \text{ bytes}$) with such a data structure;
- the storage in the GPU global memory leads to severe degradation performance, because its access time is much larger than that of the local memory of each set of SMs (Symmetric Multiprocessors).

Consequently, the use of adjacency lists, which allow a more compact graph representation, would be justified. However, its variable size for each vertex list does not allow an efficient implementation on GPUs.

To overcome such problems, we used an adapted version of the compact graph representation proposed by Harish and Narayanan [10]. Our version of this representation is composed by three vectors, V_e , E_e and L_v . Vector V_e stores vertices of a graph $G = (V, E)$, in a way that a vector index is the original vertex identification and the corresponding vector content indicates the position of its first neighbour in the adjacency vector E_e . Since the graph is undirected, it is necessary to represent each edge $(i, j) \in E$ in both adjacent lists of i and j . So, $|E_e| = 2 \cdot |E|$. Vector L_v stores the degree labels associated to each vertex of the graph. If the lists of adjacent vertices are kept sorted in E_e , then the check whether two vertices are adjacent can be performed in time $\mathcal{O}(\log \Delta)$ by a binary search.

Figure 1 illustrates this compact representation, where vertex 0 is neighbor of vertices 1 and 3, vertex 1 is neighbor of 0, 2 and 4 and so on. Considering 2 bytes for an adjacency index, this representation takes only $(|V| + |E|) \cdot 2 \cdot 2 \text{ bytes}$. This is small enough to store a graph of size

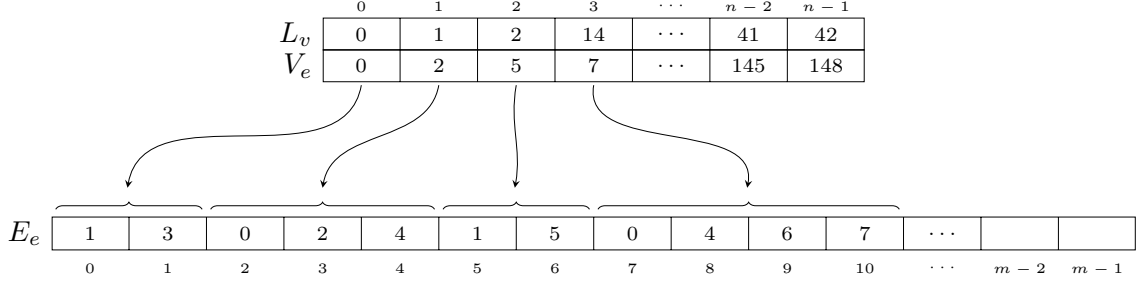


Figure 1: Compact representation of a graph.

at most 32KB in the fast local/shared memory of each SM in the majority of GPUs currently available. The search time for finding the neighbors of a vertex in this data structure is $\mathcal{O}(\Delta)$, even for dense graphs.

To allow more efficient storage of partial and complete solutions (chordless paths and chordless cycles, respectively), a map of bits was employed. A single bit is enough to indicate whether a vertex belongs to a solution because it is not important to store the vertices order in the chordless paths/cycles. This map is defined by a bi-dimensional matrix S that contains a row for each chordless path/cycle and n columns of bits, one for each vertex of the graph. In terms of bytes, the number of columns is $\lceil \frac{n}{8} \rceil$. Vertex v_j belongs to path/cycle i if, and only if, bit j of row i is 1. Despite the fact that such bitmap does not provide a vertices visit order, it depicts unambiguously each chordless path/cycle in the graph G .

In addition to the small occupied space, this data structure allows to add a vertex to a solution by a simple bitwise operation. Bit-level operations are among the least computationally expensive ones.

Figure 2 shows an example of this data structure. Row 0 contains a combination of bits that describes a chordless cycle in a graph G with $n \leq 24$. In this case, regardless the number of vertices in the graph, a path/cycle storage occupy only 3 bytes in the worst case.

However, with this matrix, it is not possible to know neither the latest vertex added to a chordless path, nor the initial or the second vertex of the path. These pieces of information are essential to the algorithm, as the last vertex is used for expanding the path, while the initial vertex allows to check whether the path forms a chordless cycle or not, and the second vertex of a path takes part of a labeling condition check.

To circumvent this problem, three auxiliary vectors, V_1, V_2 and V_L , are used. V_1 and V_2 stores the first and the second vertex of the paths and the content of their cells never changes once they were set. V_L stores the last vertex added to the chordless paths and its content is updated whenever a path is expanded. The sizes of S, V_1, V_2 and V_L have to be sufficiently large to contain information about all chordless paths that are being processed at any given moment.

Once the number of rows in each vector/matrix equals the number of chordless paths, these data structures can potentially occupy a large space in memory. Thus, they are kept in the global memory of the GPU. Even further, as we describe later, in Section 4.4, these data structures are replicated in order to speed up the processing of chordless paths.

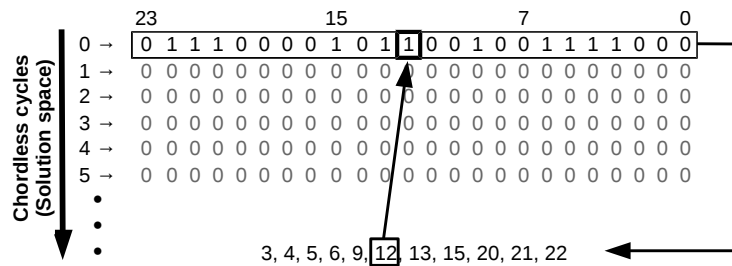


Figure 2: Solution Space, where each vertex occupies just one bit.

Now we explain the parallel implementation of our approach involving two stages.

4.3 First Stage

The first stage involves the parallelization of Lines 2 to 4 of Algorithm 1, which computes the sets C and $T(G)$. Such sets are created by selecting every vertex $u \in V$ and analyzing all pairs x and y of adjacent vertices to u , such that $\ell(u) < \ell(x) < \ell(y)$. If $(x, y) \in E$, then the triplet $\langle x, u, y \rangle$ is a simple circle and is added to C ; otherwise, $\langle x, u, y \rangle$ is a chordless path and is added to T . This process, when done sequentially, demands time $\mathcal{O}(|V| \cdot \Delta^2)$.

Our parallel approach for this stage is described in Algorithm 2. It consists of starting $M = (|V| \cdot \Delta^2(G))$ parallel threads in the GPU. Each thread j uses its unique global identifier, denoted by $gId(j)$, to compute the indices i_x, i_u and i_y of the vertices, respectively, of a triplet $\langle x, u, y \rangle$ in the compact graph representation (see Lines 2 to 4 of Algorithm 2):

$$i_u \leftarrow \left\lfloor \frac{gId(j)}{\Delta^2} \right\rfloor; \quad (1)$$

$$i_x \leftarrow \left\lfloor \frac{gId(j) - i_u \cdot \Delta^2}{\Delta^2} \right\rfloor; \quad (2)$$

$$i_y \leftarrow gId(j) \bmod \Delta; \quad (3)$$

where i_u is the index of vertex u in the vector V_e ; i_x and i_y are relative indices of x and y in the vector E_e . Index i_u ranges from 0 to $|V| - 1$, and i_x and i_y ranges from 0 to $\Delta - 1$.

Values i_x and i_y are used to determine two neighbors of the vertex u . They have to be added to the value $V_e[i_u]$ in order to obtain absolute indices in E_e . However, i_x and i_y should only be employed if they refer to valid neighbors (that is, if they are less or equal to the amount of adjacent vertices of u). Such analysis is carried out in Algorithm 2 by Lines 5 to 10. The functions *neighborsLowerBound*(u) and *neighborsUpperBound*(u) return, respectively, the absolute indices of the first and of the last neighbors of u in E_e , what allows to validate the indices i_x and i_y (Lines 8–9).

Finally, with valid vertices u, x and y , each thread tests the label condition $\ell(u) < \ell(x) < \ell(y)$. Only the threads for which this label condition is satisfied continue their execution. They check whether x is a neighbor of y and, if true, the triplet $\langle x, u, y \rangle$ is added to a set C of chordless cycles. Otherwise, the triplet is added to the set $T(G)$ of initial valid triplets (chordless paths).

As an example of the algorithm, and referring to the compact graph representation in Figure 1, to the graph in Figure 3 and consider a thread j with global unique identifier $gId(j) = 1$. That thread sets $i_u = 0, i_x = 0$ and $i_y = 1$. Next, the thread defines $k_1 = V_e[i_u] = 0, k_2 = V_e[i_u + 1] - 1 = 1, u = 0, x = E_e[k_1 + i_x] = 1$ and $y = E_e[k_1 + i_y] = 3$. Using a particular labeling, e.g., $\ell(u) = 0, \ell(x) = 1$ and $\ell(y) = 14$, the condition $\ell(x) < \ell(u) < \ell(y)$ is satisfied and $\langle x, u, y \rangle$ is an initial valid triplet. Then, since x and y are not adjacent, the triplet $\langle x, u, y \rangle$ is inserted into $T(G)$.

Lines 2 to 12 of Algorithm 2 demand constant time, while Line 13 is $\mathcal{O}(\Delta)$. Lines 14 and 16 require serialization in the index calculation in order to write $\langle x, u, y \rangle$ into C or $T(G)$ in the right position. In the worst case, $\mathcal{O}(|V| \cdot \Delta^2)$ threads may try to perform such writing operations simultaneously, but the experiments carried out show that this occurs a small amount of times for many large graphs. Moreover, this serialization is much faster than the computations done by the other lines of Algorithm 2 and it is necessary only to allocate a free memory position to write the chordless path or cycle. The writing operation, by itself, can be done in parallel.

Algorithm 2: *FindingInitialTripletsParallel*(G)

Input: Compact representation of an undirected simple graph $G = (V, E)$.

Output: Set $T(G)$ of initial chordless paths of length 3.

```

1 for each thread  $j$ , with  $j = 0, \dots, |V| \cdot \Delta^2 - 1$  do in parallel
2    $i_u \leftarrow \left\lfloor \frac{gId(j)}{\Delta^2} \right\rfloor$ 
3    $i_x \leftarrow \left\lfloor \frac{gId(j) - i_u \cdot \Delta^2}{\Delta} \right\rfloor$ 
4    $i_y \leftarrow gId(j) \bmod \Delta$ 
5    $k_1 \leftarrow neighborsLowerBound(u)$ 
6    $k_2 \leftarrow neighborsUpperBound(u)$ 
7    $u \leftarrow i_u$ 
8    $x \leftarrow (-1) \cdot (i_x > (k_2 - k_1)) + (E_e[k_1 + i_x]) \cdot (i_x \leq (k_2 - k_1))$ 
9    $y \leftarrow (-1) \cdot (i_y > (k_2 - k_1)) + (E_e[k_1 + i_y]) \cdot (i_y \leq (k_2 - k_1))$ 
10  if  $((x \neq -1) \text{ and } (y \neq -1))$  then /* both vertices must be valid */
11     $\ell(x) \leftarrow L_v(x); \ell(u) \leftarrow L_v(u); \ell(y) \leftarrow L_v(y)$ 
12    if  $((\ell(u) < \ell(x)) \text{ and } (\ell(x) < \ell(y)))$  then
13      if  $x \in Adj(y)$  then
14         $C \leftarrow C \cup \{x, u, y\}$ 
15      else
16         $T(G) \leftarrow T(G) \cup \{x, u, y\}$ 

```

4.4 Second Stage

The second stage of our approach, described in Algorithm 3, parallelizes Lines 5 to 13 of Algorithm 1. It uses all processors of the GPU in parallel for evaluating the possibility of expanding the chordless paths computed in Stage 1 (and saved in $T(G)$). This is done by allocating Δ parallel threads for every chordless path $p = \langle v_1, v_2, \dots, v_{t-1}, v_t \rangle$ in $T(G)$. Each one of the Δ threads analyzes a potential neighbor v of v_t .

Lines 5 and 6 of Algorithm 3 defines which chordless path p will be processed by thread j . Lines 7 to 10 specifies the neighbor v of v_t . If v_t has less than Δ neighbors, then there will be some exceeding threads. Such threads will fall in the condition $v = -1$, in Line 11, and they will do nothing. Finally, Lines 12 to 15 perform a task according to two cases that are similar to what we have at Stage 1:

1. If v is adjacent to v_1 but not to other vertices in $\langle v_2, \dots, v_{t-1} \rangle$, then v forms a cycle and

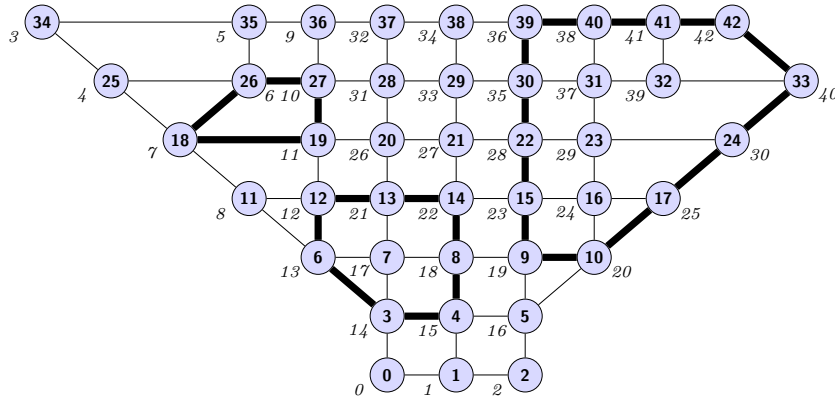


Figure 3: Part of the downtown area of the city of Goiânia, Goiás, Brazil, with degree labels near to each vertex. In highlight, three of the 9316 possible chordless cycles for the graph.

$\langle p, u \rangle$ is added to C ;

2. If v is adjacent only to v_t then v expands p and the new path $\langle p, u \rangle$ is saved in a new solution map T' ;

Some implementation details of our algorithm need to be explained. Firstly, every extended path $\langle p, u \rangle$ is added to T' instead of to T . We do that because it is faster to build a new data structure (for holding the extended chordless paths) than having to update T . In the latter case, it would be necessary to remove $\langle p \rangle$ from T in addition to adding $\langle p, u \rangle$ to this set. Secondly, we use the concept of *Persistent Threads* [8] to perform the work when there are more combinations of $|T|$ paths versus Δ neighbors than parallel processors. The loop at Line 4 does this job, by iterating the analysis for a new $p \in T$.

When all threads end their processing, they have to be restarted for working on the new set T . This task is carried out by the host process, running on the CPU, that replaces T by the recently created T' , and launches all threads again. Note, however, that we do not implement the stop condition in the host as a check $T' \neq \emptyset$. This would lead to constant communication between CPU and GPU, significantly degrading the performance of the algorithm. Instead, it is preferably a simpler approach that avoids this data transfer and that has shown to be faster: to restart all threads $|V| - 3$ times. This number of steps is sufficient for our aim, since every chordless path is increased with a new vertex of V , moved to the set C or simply discarded at each iteration of the loop. Algorithm 4 illustrates the host process.

In Algorithm 3, Line 5 and Lines 7 to 11 demand constant time. Line 6 copies a chordless path from the global GPU memory to a private thread memory. Since $\frac{|V|}{8}$ bytes are necessary to store the path, we can consider that this line takes time $\mathcal{O}(|V|)$. Lines 13 and 15 have time complexity $\mathcal{O}(\Delta \cdot t)$ for a given chordless path p under analysis, because it has to perform $\mathcal{O}(t)$ adjacency checks ($t \leq |V|$), each one of them demanding $\mathcal{O}(\log \Delta)$ verifications. Lines 14 and 16 are $\mathcal{O}(1)$, but they depend implicitly on synchronized written operations on C and T . In the worst case, $|SM| \cdot \text{MaxSMSize}$ threads would try to access one of these sets at the same time.

Therefore, the total worst-case time complexity of Algorithm 3, as a single thread execution of Line 3 of Algorithm 4, is $\frac{|T| \cdot \Delta}{|SM| \cdot \text{MaxSMSize}} \cdot (\mathcal{O}(\log \Delta \cdot t) + \mathcal{O}(|SM| \cdot \text{MaxSMSize})) = \mathcal{O}\left(\frac{|T| \cdot \Delta \cdot \log \Delta \cdot t}{|SM| \cdot \text{MaxSMSize}}\right) + \mathcal{O}(|T| \cdot \Delta)$. The second part of the time complexity is due to the synchronization process.

Consequently, Algorithm 4 has time complexity $\sum_{i=1}^{|V|} \mathcal{O}\left(\frac{|T_i| \cdot \Delta \cdot \log \Delta \cdot t_i}{|SM| \cdot \text{MaxSMSize}} + |T_i| \cdot \Delta\right)$, where $|T_i|$ is the size of the set of chordless paths in iteration i and $t_i = \Theta(i)$. Despite such a complexity seems high, the hidden constant for the synchronization step is very low and many threads fall in the case where neither C nor T are updated. Another aspect to note is that $|T_i|$ is not necessarily the same over all iterations of the loop **for** in Algorithm 4. So, the amount of computation performed can vary in each iteration. This will be illustrated later, in Section 5.

Regarding the space complexity, it is not possible to make a prediction about the amount of space that will be used. Depending on the structure of the graph under analysis, the amount of chordless cycles is potentially large.

Algorithm 3: *ExpandingChordlessPathsParallel*(G, ℓ)

Input: Compact representation of an undirected simple graph $G = (V, E)$ and list ℓ of labels.

Output: Sets T and C of chordless paths and cycles, respectively.

```
1  $globalSize \leftarrow |SM| \cdot MaxSMSize$ 
2 for each thread  $j$ , with  $j = 0, \dots, globalSize - 1$  do in parallel
3    $gId(j) \leftarrow j$ 
4   while ( $gId(j) < |T| \cdot \Delta$ ) do
5      $i_p \leftarrow \lfloor \frac{gId(j)}{\Delta} \rfloor$ 
6      $p \leftarrow getCurrentPath(T, i_p)$  //  $p$  is represented here as  $\langle v_1, v_2, \dots, v_t \rangle$ .
7      $k_1 \leftarrow neighborsLowerBound(v_t)$ 
8      $k_2 \leftarrow neighborsUpperBound(v_t)$ 
9      $i_v \leftarrow gId(j) \bmod \Delta$ 
10     $v \leftarrow -1 \cdot (i_v > (k_2 - k_1)) + (E[k_1 + i_v]) \cdot (i_v \leq (k_2 - k_1))$ 
11    if ( $v \neq -1$ ) and ( $v \notin p$ ) and ( $L_v(v) > L_v(v_2)$ ) then
12      if ( $v \in Adj(v_1)$ ) and ( $v \notin Adj(v_i), i \in \{2, \dots, t-1\}$ ) then
13         $C \leftarrow C \cup \{p, v\}$ 
14      if ( $v \notin Adj(v_i), i \in \{1, \dots, t-1\}$ ) then
15         $T' \leftarrow T' \cup \{p, v\}$ 
16     $gId(j) \leftarrow gId(j) + globalSize$ 
```

Algorithm 4: *HostProcess*(G, ℓ)

Input: Compact representation of an undirected simple graph $G = (V, E)$ and a list ℓ of labels.

Output: Set C of chordless cycles.

```
1 Create the data structures  $E_e, V_L, V_1, V, C, T$  and  $T'$ 
2  $C = \emptyset$ 
3 for  $i = 1, 2, \dots, |V| - 3$  do
4   Launch  $|SM| \cdot MaxSMSize$  threads each one running Algorithm 3
5    $T \leftarrow T'$ 
6   Wait all threads to finish
7 Return  $C$ 
```

5 Computational Experiments

Both parallel and sequential ² algorithms were coded in the C++ language and compiled using a GNU compiler (g++ version 4.8.2 with parameters “-O3 -mmodel=medium -m64 -g -W -Wall”). The parallel algorithm also used OpenCL 1.2 with the AMD Software Development Kit 2.9.1. All experiments were performed on a computer with an AMD FX-9590 Black Edition Octa Core CPU, with clock ranging from 4.7GHz to 5.0GHz, 32GB of RAM, running Ubuntu 14.04 64-bits operating system. The computer had a Radeon SAPPHIRE R9 290X Tri-X OC GPU video card, with 4GB of memory. The architecture of such a video card provides 2816 stream processing units and an enhanced engine clock of up to 1040Mhz. Its memory is clocked at 1300MHz (5.2GHz effectively).

In order to evaluate the benefits of the parallel algorithm over the sequential one, in terms of processing time to enumerate all chordless cycles, we performed intensive experiments with

²We emphasize that the sequential algorithm is the fastest one known up to now, as described in [6].

several datasets. We used twenty three graphs for the experiments, joined in three groups. The first group consists of ten graphs presented in well known databases of ecological studies [5]. These graphs, which have already been considered by Sokhn et al. [19], are formed by directed edges and represent *food webs*. For the application of such graphs in the current experiments, it was necessary to transform them into undirected *niche overlap* graphs. This was done using the definitions provided by Wilson and Watkins [25].

The second group consists of three urban traffic networks, regarding the cities of Sioux Falls, ??? and part of the downtown area of the city of Goiânia, the capital of the state of Goiás, in Brazil (illustrated in Figure 3). Finally, the last group contains well structured graphs representing a cycle, a wheel, some bipartite graphs and grids.

Table 1 presents details of each graph. It shows the name of the graph, the numbers n and m of vertices and edges respectively, and the maximum degree Δ . The remaining columns contain information produced by our algorithms. Column C_3 gives the number of cycles of length three. They are found at the first stage of the sequential and the parallel algorithms. Column $\#clc$ provides the number of chordless cycles with length greater than three, found in the graph.

The sequential and parallel algorithms were run ten times for each graph. The average running times of the ten executions are presented in the table in milliseconds. Column T_{seq} are the processing times of the sequential algorithm. The next two columns are the average times related to the parallel GPU algorithm. The first column ($T_{par-proc}$) contains only the processing time spent by the GPU kernels at the first and second stages, plus the time for the sequential degree labeling preprocessing; the second column ($T_{par-total}$) has the total time of the parallel code; this includes the processing time ($T_{par-proc}$) plus the communication time between the host and the GPU in order to transfer the graph structure and the solution set C . The last column of Table 1 is the speedup of the parallel algorithm over the sequential algorithm (given by $T_{par-total}/T_{seq}$).

Table 1: Running time to enumerate all chordless cycles on niche overlap graphs and on other well known graphs.

Name	n	m	Δ	C_3	$\#clc$	T_{seq}	$T_{par-proc}$	$T_{par-total}$	Speedup
CrystalD	24	86	14	293	0	0.333	0.182	0.622	0.536
ChesUpper	37	85	15	167	0	0.370	0.160	0.656	0.564
Narragan	35	168	22	586	0	0.548	0.197	0.709	0.773
Chesapeake	39	90	11	157	0	0.150	0.188	0.700	0.214
Michigan	39	175	27	587	0	0.614	0.197	0.698	0.879
Mondego	46	206	24	886	0	0.725	0.207	0.773	0.938
Cypwet	71	842	46	8946	0	6.417	0.258	0.892	7.196
Everglades	69	1214	56	15627	710	12.407	0.388	1.478	8.395
Mangrovedry	97	2132	80	30659	27426	102.475	1.822	6.510	15.741
Floridabay	128	3249	98	62389	85976	366.495	2.518	15.095	24.279
Goiânia	43	75	4	5	9311	39.594	0.216	3.081	12.849
SiouxFalls	24	76	5	2	176	1.339	1.138	1.812	0.739
???	???	???	???	???	???	???	???	???	???
C_{100}	100	100	2	0	1	0.149	0.165	0.770	0.193
Wheel 100	101	200	100	100	1	0.225	0.778	1.229	0.183
$K_{8,8}$	16	64	8	0	784	0.473	0.197	0.599	0.790
$K_{50,50}$	100	2500	50	0	1500625	600.661	4.867	10.391	57.805
Grid 4×10	40	66	4	0	1823	15.430	0.185	1.993	7.742
Grid 5×6	30	49	4	0	749	2.610	0.167	1.249	2.090
Grid 5×10	50	85	4	0	52620	199.132	1.982	12.718	15.658
Grid 6×6	36	60	4	0	3436	7.889	0.203	1.570	5.025
Grid 6×10	60	104	4	0	800139	2906.009	6.284	18.989	153.034
Grid 7×10	70	123	4	0	8136453	36955.470	54.840	286.212	129.119
Grid 8×10^a	80	142	4	0	71535910	427091.02	1655.147	8697.081	49.107

^aDue to high memory consumption for storing set T when processing Grid 8×10 , both the sequential and parallel algorithms were modified to not store the chordless cycles, but only to count them.

5.1 Analysis of the results

The benefits of the parallel algorithm over the sequential one depend on the nature of the graph. As we can see, the gain in speedup is, in general, proportional to the number of chordless cycles and paths, with speedups ranging from $12\times$ to $153\times$ for the most complex cases (with $|C| \geq 100.000$). When the graph had not many chordless cycles and paths, the sequential algorithm overcame the parallel GPU code.

Note however that, almost all worst cases (when the speedup was less than 1, indicating a better performance to the sequential algorithm), the most expensive activity in the parallel algorithm was the data communication between the host and the GPU device. So, when considered only the GPU kernel time (column $T_{par-proc}$), the parallel algorithm is very competitive. Furthermore, the parallel algorithm executed in less than 0.002 seconds for all non-competitive cases.

It is useful to see, as well, the evolution of sets C and T in size during an execution of the two stages of the parallel algorithm. This gives a hint about the amount of computation done by the parallel threads over time, and how much synchronization was necessary for writing on the data structures that hold such sets. Figure 4 shows this evolution for the graphs Floridabay, Mangrovedry, Grid 7×10 and Goiânia. The blue (darker) line in each chart represents the size of set T at each call of the kernels; the red (lighter) line shows the change on the size of set C . The X axis represents the results of both stages and also implies the size of all paths in the current set T . Step 1 in the chart represents the result of the first stage of our algorithm. The following steps are related to the output of each iteration (kernel call) of the second stage.

At the beginning of the computation, both C and T sets are empty. Then they are initialized by the first stage of the parallel algorithm. As the algorithm processes through the second stage, new chordless paths are created by extending smaller paths with adjacent vertices. This causes the set T to increase in size. In this case, more synchronization for writing in T and C occurs. Latter, the extension of some paths result in chordless cycles (that are then added to C) or in cycles with chords (that are just discarded). The overall process results in a wave shape for the evolution chart of T and a more soft increasing curve for the evolution chart of C .

It is interesting to note that even with a very high peak of the size of T for the graph Grid 7×10 , with 14 millions of chordless paths stored, the performance of the parallel algorithm was much superior than that of the sequential one (with a speedup of $\approx 129\times$).

A curious case was graph Mangrovedry. Many chordless cycles of size three (around 30.000) were found right at the first stage of the parallel algorithm. The second stage of the algorithm performed only seven steps (similarly to graph Floridabay), which doubles the size of C but with chordless cycles of length at most 9 (recall that the initial chordless paths have length 3, as found by the first stage of the algorithm, and they grow one edge at every iteration of the second stage).

6 Conclusions

In this paper, was presented a parallel algorithm for GPUs to enumerate all chordless cycles of a given undirected graph. The algorithm is based on a previous work done by some of the authors, which resulted in an already fast sequential algorithm for the same problem. The parallel algorithm works in two stages: in the first stage it computes an initial set of triplets and an initial set of chordless paths for expansion; in the second stage, all chordless paths are analyzed and then expanded or removed. The parallel algorithm takes advantage of the GPU architecture by distributing many tasks that are necessary in each stage to the groups of GPU processing units. A compact data structure for graph representation, distinct types of memories and the persistent thread technique were employed for allowing a more efficient usage of the GPU memory and processing power.

Experiments were done with several graphs and they showed that the benefits of the parallel algorithm depends on a large number of the chordless cycles and chordless paths in the input graph. For the graphs with more than 100.000 chordless cycles or paths, the speedup of the

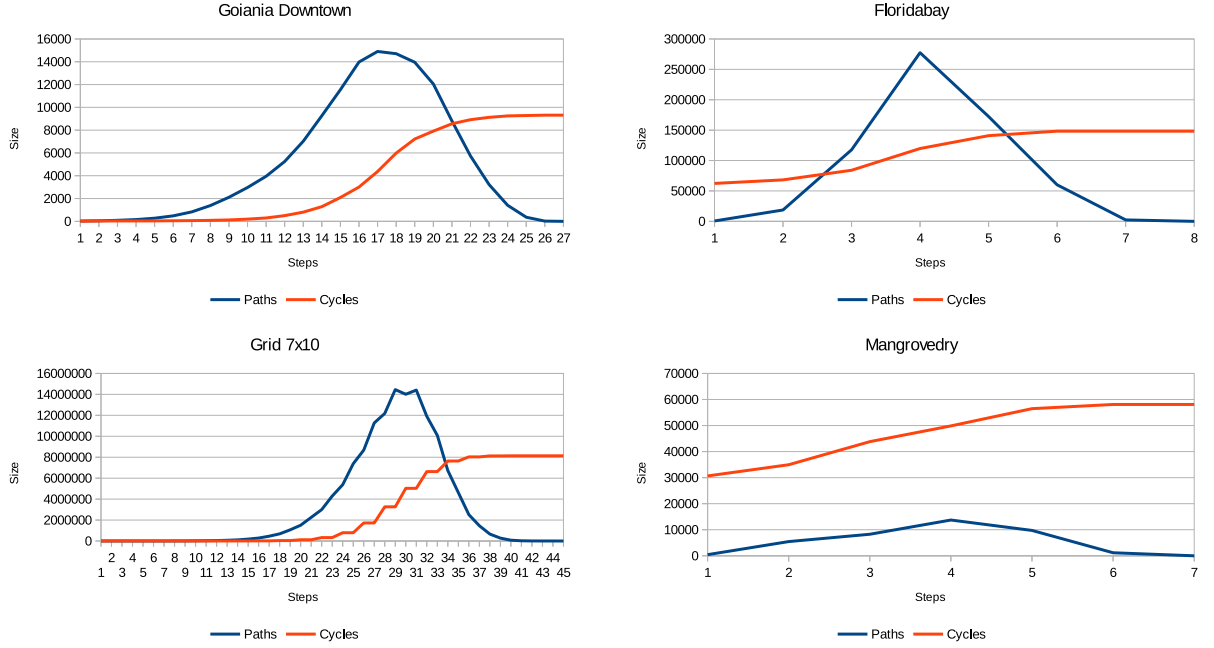


Figure 4: Sizes of T and C for four graphs.

parallel algorithm over the sequential one was from ≈ 12 to 153 times. The cases for which the parallel algorithm was worse (took longer than the sequential algorithm) were the ones with very few chordless cycles and most of the exceeding time was spent with data transfer between the CPU and the GPU. For those base cases, our implementation still took less than 0.002 seconds to find all chordless cycles.

Up to our knowledge, this is the first parallel GPU-based algorithm for the problem of enumerating all chordless cycles. Note, however, that memory size on a GPU is still a restricting factor since the data structures cannot be larger than the maximum supported texture size. Such hardware constraints limit the size of the problems and solutions that can be dealt with by the GPUs. Thus, as a future work, we are planning to develop a new data transportation protocol between the ordinary RAM memory and the GPU memory in order to open space when necessary and allow to enumerate chordless cycles for much larger datasets. We are also implementing a parallel algorithm for computing the degree labeling. Deleting a vertex during such a computation can lead to a major change in the graph (the decrease of one unit of the degree of every adjacent vertex), what indicates that the labeling process has an inherent sequential nature. However, one could update the degree of all vertices in parallel in constant time using $n \cdot \Delta$ processors. Then, the smallest degree can be found through a parallel reduction in time $\mathcal{O}(\log(n))$ with n threads. Repeating this process $n - 1$ times provides the desirable result with total time $\mathcal{O}(n \log(n))$.

Acknowledgement

We thank the Brazilian research supporting agencies CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) for providing Ph.D. scholarships to the first and the second authors, respectively.

References

- [1] Birmelé, E., Ferreira, R., Grossi, R., Marino, A., Pisanti, N., Rizzi, R., and Sacomoto, G. (2013). Optimal listing of cycles and *st*-paths in undirected graphs. In *Proceedings of SODA '13*, pages 1884–1896. SIAM.

- [2] Bisdorff, R. (2010). On enumerating chordless circuits in directed graphs. Available at <http://sma.uni.lu/bisdorff/ChordlessCircuits/documents/chordlessCircuits.pdf>.
- [3] Bondy, J. A. and Murty, U. S. R. (1976). *Graph theory with applications*, volume 6. Macmillan London.
- [4] Chandrasekharan, N., Laskshmanan, V., and Medidi, M. (1993). Efficient parallel algorithms for finding chordless cycles in graphs. *Parallel Process. Lett.*, 3(2):165–170.
- [5] Cohen, J. (1978). *Food Webs and Niche Space*. Princeton University Press.
- [6] Dias, E. S., Castonguay, D., Longo, H. J., and Jradi, W. A. R. (2013). Efficient enumeration of all chordless cycles in graphs. *CoRR*, abs/1309.1051.
- [7] Dogrusöz, U. and Krishnamoorthy, M. (1995). Cycle vector space algorithms for enumerating all cycles of a planar graph. *J. Parallel Algo. Appl.*, pages 1–14.
- [8] Gupta, K., Stuart, J. A., and Owens, J. D. (2012). A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Innovative Parallel Computing (InPar)*, 2012, pages 1–14. IEEE.
- [9] Haas, R. and Hoffmann, M. (2006). Chordless paths through three vertices. *Theor. Comput. Sci.*, 351:360–371.
- [10] Harish, P. and Narayanan, P. (2007). Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of HiPC '07*, pages 197–208. Springer-Verlag.
- [11] Kapoor, S. and Ramesh, H. (2000). An algorithm for enumerating all spanning trees of a directed graph. *Algorithmica*, 27(2):120–130.
- [12] Lefohn, A. E., Sengupta, S., Kniss, J., Strzodka, R., and Owens, J. D. (2006). Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graphic (TOG)*, 25(1):60–99.
- [13] Liu, H. and Wang, J. (2006). A new way to enumerate cycles in graph. In *Proceedings of AICT/ICIW '2006*, pages 57–59.
- [14] Loizou, G. and Thanisch, P. (1982). Enumerating the cycles of a digraph: A new preprocessing strategy. *Inform. Sci.*, 27:163–182.
- [15] Makino, K. and Uno, T. (2004). New algorithms for enumerating all maximal cliques. *Lecture Notes in Comput. Sci., SWAT 2004*, 3111:260–272.
- [16] Read, R. and Tarjan, R. (1975). Bounds on backtrack algorithms for listing cycles, paths and spanning trees. *Networks*, 5:237–252.
- [17] Sankar, K. and Sarad, A. V. (2007). A time and memory efficient way to enumerate cycles in a graph. In *Proceedings of ICIAS '2007*, pages 498–500. IEEE.
- [18] Satoh, H., Koshino, H., Uno, T., Koichi, S., Iwata, S., and Nakata, T. (2005). Effective consideration of ring structures in cast/cnmr for highly accurate ^{13}C {NMR} chemical shift prediction. *Tetrahedron*, 61(31):7431 – 7437.
- [19] Sokhn, N., Baltensperger, R., Bersier, L., Hennebert, J., and Nitsche, U. (2013). Identification of chordless cycle in ecological networks. *LNICST, COMPLEX 2012*, 126:316–324.
- [20] Tarjan, R. E. (1972). Enumeration of the elementary circuits of a directed graph. Technical report, Computer Science Technical Reports – Cornell University. Available at <http://ecommons.library.cornell.edu/handle/1813/5941>.
- [21] Tomita, E., Tanaka, A., and Takahashi, H. (2006). The worst-case time complexity for generating all maximal cliques and computational experiments. *Theo. Comp. Sci.*, 363:28–42.

- [22] Uno, T. and Satoh, H. (2014). An efficient algorithm for enumerating chordless cycles and chordless paths. Available at <http://arxiv.org/pdf/1404.7610v1.pdf>.
- [23] Valiant, L. (1979). The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421.
- [24] Wild, M. (2008). Generating all cycles, chordless cycles, and hamiltonian cycles with the principle of exclusion. *J. Discrete Algorithms*, 6(1):93–102.
- [25] Wilson, R. J. and Watkins, J. J. (1990). *Graphs: An Introductory Approach*. Wiley, Michigan University.